Basics of the Java Persistence API – Understanding the Entity Manager

Applies to:

SAP NetWeaver Application Server, Java EE 5 Edition

Summary

Applications that use the new Java Persistence API (JPA) interact with the entity manager – the central interface of JPA – to store data in and retrieve data from a relational database. This article explains the purpose and usage of the different methods in the entity manager interface and elucidates the most fundamental ideas behind it. Special attention is paid to potential programming pitfalls intrinsic to the entity manager API as well as to the specific behavior of SAP JPA 1.0, the JPA implementation delivered with SAP NetWeaver Application Server, Java EE 5 Edition.

Author(s): Sabine Heider

Company: SAP AG

Created on: 15 January 2007

Author Bio

Sabine Heider, SAP AG

Sabine is a member of the SAP NetWeaver Application Server Java development team, focusing on Java persistence and the different persistence technologies. Together with her colleagues, she has developed SAP JPA 1.0, which is part of SAP NetWeaver Application Server, Java EE 5 Edition. She joined SAP in 1997.

Table of Contents

The Entity Manager Interface

The main idea behind the Java Platform, Enterprise Edition 5 (Java EE 5) is ease of development. You'll find this concept throughout all technologies that Java EE 5 comprises, but some of the most striking improvements have certainly been achieved in the area of object-relational persistence where a completely new API has been brought up. The Java Persistence API (JPA) is a lightweight POJO ("plain old Java object") persistence model that is both powerful and easy to use. In our previous article (Getting Started with Java Persistence API and SAP JPA 1.0), we gave an example of a simple enterprise application that makes use of JPA to persist an entity to a relational database, to find it by its primary key, and to delete it from the database. You saw that for all these operations, we interacted with the so-called *entity manager*, the key concept of the Java Persistence API, which is modeled by the Java interface `javax.persistence.EntityManager`.

In this article, we will introduce you to the `EntityManager` interface and give you useful hints how to work efficiently with this API. We will explain the most common methods in detail, while others that address more advanced features of JPA are only described in principle. Further articles in our series on JPA will give more insight there. The area of queries is also sketched only roughly here – a comprehensive article that covers that topic in depth will follow soon.

Note that this article does not intend to replace the official API documentation (aka javadoc, see section *Additional Resources*), which should be your first reference when using the Java Persistence API. Our goal is to comment on the entity manager interface and to provide some additional information where the javadoc remains rather short or requires a good amount of background knowledge about JPA.

Summary of Entity Manager Operations

Table 1 summarizes the operations defined by the interface `javax.persistence.EntityManager` as described by the javadoc. We will explain you these methods in detail in the subsequent sections.

`javax.persistence.EntityManager`	
Basic Operations	
`void`	**`persist(Object entity)`** Make an entity instance managed and persistent.
`<T> T`	**`find(Class<T> entityClass, Object primaryKey)`** Find by primary key.
`<T> T`	**`merge(T entity)`** Merge the state of the given entity into the current persistence context.
`void`	**`remove(Object entity)`** Remove the entity instance.
Basic Query Operations	
`Query`	**`createNamedQuery(String name)`** Create an instance of Query for executing a named query (in the Java Persistence query language or in native SQL).
`Query`	**`createQuery(String qlString)`** Create an instance of Query for executing a Java Persistence query language statement.

Advanced Query Operations	
`Query`	**`createNativeQuery(String sqlString, Class resultClass)`** Create an instance of Query for executing a native SQL query.
`Query`	**`createNativeQuery(String sqlString, String resultSetMapping)`** Create an instance of Query for executing a native SQL query.
`Query`	**`createNativeQuery(String sqlString)`** Create an instance of Query for executing a native SQL statement, e.g., for update or delete.
`FlushModeType`	**`getFlushMode()`** Get the flush mode that applies to all objects contained in the persistence context.
`void`	**`setFlushMode(FlushModeType flushMode)`** Set the flush mode that applies to all objects contained in the persistence context.
Synchronization with the Database	
`void`	**`flush()`** Synchronize the persistence context to the underlying database.
`void`	**`refresh(Object entity)`** Refresh the state of the instance from the database, overwriting changes made to the entity, if any.
Contents of the Persistence Context	
`boolean`	**`contains(Object entity)`** Check if the instance belongs to the current persistence context.
`void`	**`clear()`** Clear the persistence context, causing all managed entities to become detached.
Lifecycle of the Entity Manager	
`void`	**`close()`** Close an application-managed EntityManager.
`boolean`	**`isOpen()`** Determine whether the EntityManager is open.
Concurrency Control	
`void`	**`lock(Object entity, LockModeType lockMode)`** Set the lock mode for an entity object contained in the persistence context.

Transactions	
`EntityTransaction`	**`getTransaction()`** Returns the resource-level transaction object.
`void`	**`joinTransaction()`** Indicate to the EntityManager that a JTA transaction is active.
Others	
`<T> T`	**`getReference(Class<T> entityClass, Object primaryKey)`** Get an instance, whose state may be lazily fetched.
`Object`	**`getDelegate()`** Return the underlying provider object for the EntityManager, if available.

Table 1: The Interface `javax.persistence.EntityManager`

How the Entity Manager Works Internally – The Persistence Context

Before we go into the details of the entity manager API, let's introduce a concept that is crucial for a solid understanding of JPA: the persistence context. It will not only help you to understand the behavior of particular API methods, it will also clarify the lifecycle of an entity.

As you know, the entity manager manages persistent entities on behalf of the application and carries out all operations that are necessary to synchronize the entities' state with the database. Internally, the entity manager maintains a set of entity instances that are currently under its control – the so-called *persistence context*. Only entities that are contained in the persistence context are managed by the entity manager, which means that changes to them will eventually be reflected in the database. Entities that are not contained in the persistence context are just regular Java objects without special characteristics.

The entity manager updates or consults the persistence context whenever you call a method of the `EntityManager` interface. When you call the `persist` method, for example, the entity that you pass as an argument will be added to the persistence context (if not already present there). Similarly, when you obtain an entity from the entity manager as a result of the `find` method, the entity manager checks at first whether the requested entity is already present in the persistence context. If not, the entity manager reads the entity from the database and adds it to the persistence context. Note that the persistence context holds at most one entity instance per primary key.

Scope of the Persistence Context

Entities that are controlled by an entity manager can become unmanaged again in two ways: by an explicit entity manager call that clears the persistence context (see section *Contents of the Persistence Context – The Methods `contains` and `clear`* later in this article) or automatically when the persistence context is closed. Since you don't have direct access to the persistence context, the latter is driven by the container rather than the application. To avoid surprises, it is therefore worth investigating the topic in order to know when this happens exactly.

JPA defines two types of persistence contexts that have different lifecycles: transaction-scoped persistence contexts and extended persistence contexts. A *transaction-scoped persistence context* is exactly what its name suggests: It lives as long as the underlying transaction. The entity manager automatically creates a new persistence context upon the first operation performed within a transaction, reuses the same persistence context for multiple operations within the same transaction, and finally closes the persistence context after the transaction has ended (usually with commit). Within the following transaction, the entity manager starts with a new, initially empty persistence context.

Entities Become Unmanaged at the End of the Transaction

An immediate consequence of the limited lifetime of a transaction-scoped persistence context is that all previously managed entities become unmanaged at the end of the transaction. Although obvious and intended, this behavior is also a common programming pitfall for those new to JPA. Especially when encapsulating JPA in a session bean with container-managed transaction demarcation, it's easy to forget about the transactional scope of the persistence context. The container opens and closes the transaction automatically depending on the transaction attributes you declared – so if your business method returns any entities, they may already be unmanaged.

Transaction-scoped persistence contexts are appropriate for most use cases and are frequently used within business applications. Occasionally, however, you might want the entity manager to control your entities even beyond the end of the transaction. For an application that accesses and modifies entities in a wizard-like manner over several dialog steps, for example, it might be convenient to work with entities that remain managed outside of a transaction. This is what the *extended persistence context* is for. An extended persistence context lives as long as the entity manager that uses it and may therefore span multiple transactions. Once an entity has been added to the persistence context, it remains managed until the entity manager itself is closed or the persistence context is cleared explicitly. Any change that you apply to a managed entity, even outside of a transaction, will eventually be reflected in the database. The entity manager keeps track of those changes and uses a later transaction for the synchronization with the database.

A thorough discussion of extended persistence contexts including typical use cases goes beyond the scope of this document. A separate article later in this series will go more into details.

Basic Operations

To start with, let's have a look at the most fundamental entity manager operations: `persist`, `find`, `merge`, and `remove`. Together with the basic query operations that we are going to introduce in the next section, they provide enough functionality for most JPA applications. The remaining methods in the entity manager interface address mainly particular scenarios or special use cases – so it may take a while until you actually need any of them.

Persisting an Entity – The `persist` Method

The `persist` method adds a new entity instance to the persistence context, thus scheduling it for insert into the database. With a transaction-scoped persistence context, an active transaction is required, of course, while there is no such restriction if an extended persistence context is used.

The following code snippet demonstrates how to use the `persist` method in practice:

```
// set up a new entity instance
Employee employee = new Employee(10);
employee.setName("Miller");
employee.setSalary(70000);

// put it under the management of the entity manager
em.persist(employee);

// changes to the data are still possible
em.setSalary(75000);
```

As a first step, we create a regular Java instance of type `Employee`. By calling the `persist` method, the entity is put under control of the entity manager and remains managed until the persistence context ends, that is usually until the end of the transaction. Up to that time, we may still change the entity's data and can

be sure that the latest state will be reflected in the database after commit. That's exactly the essence of the entity being managed!

Database Operations are Performed at an Undefined Point in Time

JPA leaves it up to the implementation to decide at which point in time the insert into the database (or more generally, the database operation that corresponds to an entity manager call) is actually performed. Most implementations try to delay the database operations as long as possible, typically until commit time, to reduce the impact of potential database locks – but you should by no means rely on that behavior. It may not only vary from vendor to vendor, even one and the same vendor might decide to change the behavior in a later version of its product.

SAP JPA 1.0 delays the execution of SQL statements. Entities for which the `persist` method is called are inserted only when the entire persistence context is synchronized with the database, i.e. usually at commit time.

In the regular case, you pass a new entity to the `persist` method, that is an entity for which there is no corresponding entry in the database yet. If an entity with the same primary key exists already, however, the operation will eventually fail, but it depends on the particular implementation how exactly. JPA allows two different behaviors: either the `persist` method fails immediately with an `EntityExistsException`, or `persist` works but later the commit of the transaction fails.

If you persist an entity with SAP JPA 1.0 and an entity with the same primary key exists already in the database, the `persist` operation works, but synchronizing the persistence context with the database later on, usually during commit, fails.

Finding an Entity by Its Primary Key – The `find` Method

A very common task when writing a JPA based application is to retrieve an entity via its primary key. You will usually use the `find` method for this purpose:

```
<T> T find(Class<T> entityClass, Object primaryKey)
```

If you are already familiar with Java SE 5, you understand the signature: The method returns an instance of the class that you pass as a first argument, so casting is not required. See the following example:

```
Employee employee = em.find(Employee.class, Integer.valueOf(10));
if (employee != null) {
    // do something
}
```

The `find` method returns the entity with the primary key 10 or `null` if there is no such entity. When you use the `find` method in your application, remember to check its return value for `null` before accessing it!

If there is a transaction active while you execute `find` (or if an extended persistence context is used), the method returns a managed entity. On the other hand, `find` is a read-only operation, so you do not necessarily need an active transaction. But don't forget that in the typical configuration (using a transaction-scoped persistence context), there is no persistence context available outside of a transaction. Correspondingly, the `find` method returns an entity that is not managed. You can use the entity, read its data, even change its data – but don't expect those changes to be written to the database automatically.

Don't mistake the behavior of the `find` method: The requested entity is not necessarily read from the database. The entity manager will consult the underlying persistence context (if present) first. If the entity is

available there, the entity manager returns the managed entity instance, whose data might have been changed already compared to the version in the database. Only if the entity is not available in the persistence context, it is actually from the database.

Start a Transaction Only When Needed

It is generally good programming practice to use transactions consciously. Maintaining a transaction means a considerable effort for the application server, which is just wasted if you simply want to read some data, for example. So open a transaction only if you really need one!

In case you use session beans to encapsulate your business logic, you usually don't start the transactions yourself but rely on the application server to do this on behalf of you. Still you can declare that a certain business method does not necessarily need a transaction by annotating the business method with `@TransactionAttribute(SUPPORTS)`*. That way, the container will use an existing transaction if there is one already, but it will not start a new one if there is not.*

Merging a Detached Entity – The `merge` Method

If the application references an entity that exists in the database but is not (or not any more) managed by the entity manager, we say the entity is *detached*. With a transaction-scoped persistence context, this is quite a common situation: Any managed entity instance becomes detached at the end of the transaction, while an entity returned by the `find` method or a query outside of a transaction is immediately detached. Since a detached entity is just a regular Java object, you can access and change its data independent from the entity manager, for example you can display it on a web page and have the user manipulate its data. To bring the potentially modified object back under control of the entity manager (and thus finally to the database), you have to call the `merge` method:

```
<T> T merge(T entity)
```

Like the `persist` method, `merge` requires an active transaction when called on an entity manager with a transaction-scoped persistence context.

The `merge` operation involves two instances of the same entity – the detached instance that you pass as an argument, and the managed version of the entity available in the persistence context. If the persistence context doesn't contain the corresponding entity yet, the entity manager reads it from the database in a first step. The entity manager then copies the data of the detached entity that you passed into the managed entity instance. The return value of the `merge` operation is the managed version of the entity with your data copied into it. The code snippet below illustrates the principle usage of the `merge` method.

```
// -> assume we are outside a transaction

// get a detached entity
Employee detachedEmployee = em.find(Employee.class, Integer.valueOf(10));
// change entity while detached
detachedEmployee.setSalary(80000);

[…]
// -> assume a transaction has been started in the meantime

// merge changed entity back
Employee mergedEmployee = em.merge(detachedEmployee);

// the result of merge is a managed entity with the changed data, i.e.
// mergedEmployee.getSalary() == 80000;
```


Removing an Entity – The `remove` Method

The `remove` method allows you to remove an existing entity from the database. It requires an active transaction when called on an entity manager with a transaction-scoped persistence context.

The `remove` method takes the entity that you want to remove as an argument. But notice: The entity has to be managed already! That means, if the entity is not available in the persistence context as a result of a previous action, you have to read it from the database just for the purpose of deleting it afterwards. See the following code as an example:

```
// get a managed entity
Employee employee = em.find(Employee.class, Integer.valueOf(10));
if (employee != null) {
    em.remove(employee);
}
```

Although the performance impact for a single `remove` operation is indeed considerable, a real-life application is usually less affected than you might expect. In the first instance, removing entities is usually not that common. Many applications read and modify data much more often than they create or remove them, so the slower remove operation does not lead to measurable effects. In exceptional cases, where you indeed experience a performance degradation, you can bypass the entity manager and execute a special type of "query" to delete entities directly on the database. Alternatively, using the `getReference` method instead of `find` might lead to slightly better results with some JPA implementations (see section *Other Methods – `getReference` and `getDelegate`* for details).

Basic Query Operations

The `find` method allows you to locate an entity by its primary key, which convenient and frequently used, but not sufficient for most applications. Often you want to search for entities based on their data. Thus you need a way to express your search criteria and a mechanism to execute a request and retrieve the results. This is what the Java Persistence Query Language (JPQL) and the query object are for.

There is a lot to say about queries and the query language, and we will dedicate a separate article to that topic. So let's just have a look at the overall concept and a few sample queries.

JPQL Examples

If you look at sample JPQL queries, they will most likely remind you of SQL queries. The main difference is that you query the application model, i.e. the entities, rather than any database tables.

Consider for example the following query, which selects all instances of type Employee:

```
SELECT e FROM Employee e
```

In the `FROM` clause, you state that you want to select from the entity `Employee`, and you give it the alias `e` (also called identification variable) by which you refer to the `Employee` in the other parts of the JPQL query. In the `SELECT` clause, you declare that you want to select instances of type `e`, i.e. `Employee`, so the result of this query is a list of employee instances.

If you want to limit your result list to entities that meet some given search criteria, you need to add a `WHERE` clause to your query. Again, it's similar to the `WHERE` clause in SQL, but you refer to your entity and its persistent fields or properties in a Java-like manner. For example, assume that your entity `Employee` has a persistent field called `lastName`. To select only those entity instances where `lastName` has a given value, you will use a query like:

```
SELECT e FROM Employee e WHERE e.lastName = 'Miller'
```

or rather, to be able to pass the particular value later upon query execution:


```
SELECT e FROM Employee e WHERE e.lastName = :name
```

Again, you stay completely inside your application's object model. Dealing with the object-relational side, finding out the name of the database table to which the entity is mapped, determining the name of the columns that correspond to the persistent fields, and finally translating the query into proper SQL – all that is the task of the JPA runtime.

Creating a Query Object for a Static Query – The `createNamedQuery` Method

A query in JPA is implemented as a separate object of type `javax.persistence.Query`, which you create via a factory method of the entity manager. You can choose between two types of JPQL queries: static queries, which are also called named queries, and dynamic queries.

Static queries are defined at design time of the application using the `@NamedQuery` annotation, which can be placed on the class definition of any entity within the application model. Thus, you give the query a name by which it can be referred to in the application and you specify the corresponding JPQL query string. See the following example:

```
@NamedQuery(name="findAllEmployees", query="SELECT e FROM Employee e")
@Entity
public class Employee {
    […]
}
```

To define multiple static queries on the same entity class, place the `@NamedQuery` annotations inside a `@NamedQueries` annotation (note the plural!) as follows:

```
@NamedQueries({
    @NamedQuery(name="findAllEmployees",
            query="SELECT e FROM Employee e"),
    @NamedQuery(name="findEmployeeByLastName",
            query="SELECT e FROM Employee e WHERE e.lastName = :name")
})
@Entity
public class Employee {
    […]
}
```

Remember the name that you give to a static query, because that's how you refer to it in the application code when you create a query object for execution. Simply call the `createNamedQuery` method of the entity manager and pass the name of the query as an argument, for example:

```
Query query = em.createNamedQuery("findAllEmployees");
```

Creating a Query Object for a Dynamic Query – The `createQuery` Method

With static queries you will most likely be able to cover a great part of the queries within your application. Occasionally, however, you may be in a situation where you simply cannot write down the query in advance. This is what dynamic queries are for. Dynamic queries are defined using the `createQuery` method of the entity manager, passing the JPQL query string as an argument:

```
String selectFrom = "SELECT e FROM Employee e";
String whereClause = getWhereClause();
Query query = em.createQuery(selectFrom + whereClause);
```

Use Static (Named) Queries Wherever Possible

Static queries should be your query type of choice wherever adequate. Not only do

> *they give the JPA runtime the chance to translate the queries in advance and cache the results, thus potentially improving performance, they also organize your queries naturally, and they make it possible to detect query errors during design time or deploy time. Static queries in combination with query parameters still give you a certain degree of flexibility.*

Executing a Query

Although this article focuses on the `EntityManager` API rather than the `Query` interface, let's shortly look at an example how to finally execute the query that we just created:

```
Query query = em.createNamedQuery("findEmployeeByLastName");
query.setParameter("name", "Miller");
List result = query.getResultList();
```

Static and dynamic queries differ in the way they are created, but you execute them in exactly the same way. Here, we use the static query called *findEmployeeByLastName*, which corresponds to the JPQL query string `"SELECT e FROM Employee e WHERE e.lastName = :name"`. Note that this query is parameterized, so before we can execute the query, we have to bind a value to the parameter `:name`. This is done with the method `setParameter` on the query object. Once the parameter is set, we can then execute the query by calling the `getResultList` method.

The `setParameter` method returns the query object itself, which allows you to chain the calls, so you will often see a query being executed in the following way instead:

```
List result = em.createNamedQuery("findEmployeeByLastName")
                .setParameter("name", "Miller")
                .getResultList();
```

Advanced Query Operations

Creating a Query Object for a Native Query – The `createNativeQuery` Methods

In the area of queries, JPA faces a dilemma: On the one hand, JPA tries to be portable across different database platforms. Its query language JPQL therefore offers only a limited feature set that should be covered – in principle – by any major database platform. On the other hand, however, not all applications intend to be portable at all. Often an application is written for a particular database, which may be given by an external constraint or chosen intentionally, and the application wants or needs to make use of whatever the particular database offers regarding functionality and performance. JPA accepts that as a requirement and allows you to create queries based on vendor specific SQL strings in addition to the normal JPQL queries.

There are different methods in the entity manager interface for that purpose, which you choose depending on the return values you expect from the query. The first flavor

```
Query createNativeQuery(String sqlString, Class resultClass)
```

uses a default mapping of the result set. You can use it whenever you return instances of a single entity class only, and if the names of the column aliases in your `SELECT` statement are equal to the names of the persistent field or properties. See the following example:

```
List result = em.createNativeQuery(
                        "SELECT id, name, salary FROM EMPLOYEE_TABLE",
                        Employee.class)
               .getResultList();
```


In this case, the entity `Employee` consists of the persistent fields or properties `id`, `name`, and `salary`. The JPA runtime executes the given SQL statement, instantiates an `Employee` for each row in the result set, and fills the selected items into the fields or properties of corresponding name.

Obviously, the default result set mapping is limited and cannot always be used. In those cases, you have to declare the result set mapping explicitly using the `@SqlResultSetMapping` annotation. Explaining the annotation in detail goes far beyond the scope of this article, so we postpone that to a later article. Here, it's enough to know that you give a name to the result set mapping by which you can refer to it when creating a native query:

```
Query createNativeQuery(String sqlString, String resultSetMapping)
```

The third method to create a native query takes only the SQL string as an argument:

```
Query createNativeQuery(String sqlString)
```

Obviously, there is no information on the expected return value or mapping, so this method can only be used with SQL statements that don't expect a result, that is, with `INSERT` and `UPDATE` statements.

Creating a Query Object for a Static Native Query

The `createNativeQuery` methods allow you to create a query object for a dynamic native query. Static (or named) native queries are available, too, and should be chosen wherever possible. Note that there is no dedicated API method to create a query object for a static native query – the method `createNamedQuery` can be used for both JPQL and native queries. To declare a static native query, however, you have to annotate your entity class with `@NamedNativeQuery` instead.

Automatic Database Synchronization Before Query Execution – The Flush Mode

As we have already mentioned, most JPA implementations try to minimize the interaction with the database for performance reasons and delay all database operations as far as possible, ideally until commit time. That means, on the other hand, that the entities in the persistence context may have been changed compared to the version in the database, so that the database does not reflect the latest state. This has to be taken into account when executing queries, as they are carried out by the database engine, not the entity manager! The JPA implementation has to take care that the results of the query are based on the latest state of entities, whether or not present in the database already. It is up to the particular implementation how to achieve that, but you should be aware of the implementation choice of your provider. It might have a negative impact on the performance of the application, especially when queries are executed frequently.

SAP JPA 1.0 synchronizes the persistence context with the database before query execution, that is, any changed data (if there is such) is written to the database.

As a default, this behavior is absolutely reasonable, as it leads to consistent query results and aims towards functionality and ease of use rather than performance. Fortunately, however, JPA lets you avoid the automatic synchronization with the database on a per-query base by specifying the so-called flush mode. If performance is an issue, you may change the flush mode for a query from `FlushModeType.AUTO` (the default, leading to the behavior we just described) to `FlushModeType.COMMIT`, which allows the JPA implementation to omit the synchronization of the query results:

```
List result = em.createNamedQuery("findAllEmployees")
                .setFlushMode(FlushModeType.COMMIT)
                .getResultList();
```


Note that the outcome of such a query is potentially wrong or outdated, so make sure you use that option only if your application logic allows such inaccuracies.

The flush mode can not only be set on the query object, but also on the entity manager:

```
em.setFlushMode(FlushModeType.COMMIT);
```

It allows you to override the default behavior for all queries on which no explicit flush mode has been set. The corresponding method `getFlushMode` is used to determine the flush mode of the entity manager.

Set the Flush Mode Only on Query Level

Although the possibility to set the flush mode globally on entity manager level might look convenient, it is a potential pitfall and we strongly recommend not to use it. The decision to omit data synchronization at the cost of potentially incorrect results should be taken very consciously and on a per-query base. Setting a default value that can easily be overlooked, especially when adding functionality at a later point in time, is dangerous and error-prone.

Furthermore, the flush mode is a property of the persistence context rather than the entity manager. When using a transaction-scoped persistence context, you would have to set the flush mode in every transaction to achieve the desired effect.

Advanced Operations

Background Knowledge: Entity Manager Types and Transaction Types

Container-Managed Entity Manager Versus Application-Managed Entity Manager

When using JPA in the context of a Java EE 5 application, you usually obtain the entity manager via dependency injection using the `@PersistenceContext` annotation or via a JNDI lookup. Either way, you do not instantiate the entity manager yourself, but have the container do that on behalf of you. Correspondingly, the container detects the situation when a particular entity manager instance is no longer needed and closes it automatically. As the lifecycle of the entity manager is thus controlled by the container, we speak of a *container-managed entity manager*.

Alternatively, you can choose to control the lifecycle of the entity manager yourself using a so-called *application-managed entity manager*. In this case, the entity manager is created using an entity manager factory, and it must be closed explicitly by calling the `close` method on the entity manager instance. Application-managed entity managers are particularly important for Java SE applications, where they are in fact the only available option. However, even in Java EE environments there are use cases where an application-managed entity managers may be suitable.

Table 2 lists the possible combinations of entity manager type and persistence context type. The most typical configuration for Java EE environments, that is a container-managed entity manager with a transaction-scoped persistence context, is highlighted. In Java SE environments, only application-managed entity managers are available. Note that an application-managed entity manager is always related with an extended persistence context.

		Entity Manager Type	
		Container-Managed (Java EE only)	**Application-Managed**
Persistence Context Type	**Transaction-Scoped**	☑	(n.a.)
	Extended	☑	☑

Table 2: Possible Combinations of Entity Manager Type and Persistence Context Type

JTA Transaction Versus Resource-Local Transaction

When we speak of transactions in the context of JPA, we often (but not always) mean transactions according to the Java Transaction API (JTA), managed by the transaction manager of the Java EE application server. Primarily for use outside of the server in the Java SE case, however, JPA supports also a second type of transactions, the so-called *resource-local transactions*. A resource-local transaction is a pure database transaction that can be accessed and controlled via the `javax.persistence.EntityTransaction` interface.

You decide on a transaction type statically for your entire persistence unit. It can be specified explicitly in the `persistence.xml` file; if no transaction type is provided there, JTA transactions are assumed.

Table 3 lists the possible combinations of entity manager type and transaction type. The most typical configuration for Java EE environments, that is a container-managed entity manager, is highlighted. In this case, JTA is required as the underlying transaction type. In Java SE environments, only application-managed entity managers with resource-local transactions are available.

		Entity Manager Type	
		Container-Managed (Java EE only)	**Application-Managed**
Transaction Type	**JTA (Java EE)**	☑	☑
	Resource-Local	(n.a.)	☑

Table 3: Possible Combinations of Entity Manager Type and Transaction Type

Synchronization with the Database – The Methods `flush` and `refresh`

For most applications, the automatic data synchronization that JPA supports is just sufficient. Although you don't know at which point in time the JPA implementation actually writes down the changed data to the database, you can be sure that it happens, often right before commit of the transaction. If you have a query that relies on current data in the database, a proper setting of the flush mode achieves that automatically without further action (see section *Automatic Database Synchronization Before Query Execution – The Flush Mode*). In rare cases, however, you may want to control data synchronization explicitly and force the JPA implementation to write changes to the database. This is what the `flush` method is for. You may call the `flush` method only if there is an active transaction.

Use Explicit Synchronization Consciously

It is strongly advised to use the `flush` method only if actually needed. The database operations may cause locks in the database, which will be kept until the end of the transaction and which may cause performance degradations. Many JPA implementations try to optimize their automatic data synchronization process in order to reduce these effects – calling `flush` explicitly interferes with those attempts.

The `refresh` method is used for data synchronization in the opposite direction. It allows to overwrite the current state of a managed entity instance with the data as it is present in the database.

There are two typical use cases where you might want to use the `refresh` method, typically in combination with an extended persistence context: to read data that might have been changed by another transaction, or to undo changes that you have carried out in memory only. See the following code snippet as an example for an undo operation based on the `refresh` method:

```
// -> assume an extended persistence context outside a transaction

Employee employee = em.find(Employee.class, new Integer(10));
int originalSalary = employee.getSalary();
// update the value
employee.setSalary(salary * 2);
// -> changed only in memory as we are outside a transaction

// undo changes
em.refresh(employee);
assert employee.getSalary() == originalSalary;
```

Note that we assume that the code is executed outside a transaction. Within the scope of an active transaction, the example above could potentially fail on some JPA providers. The reason is that our implementation of the undo functionality above relies on the fact that the change of the entity's data has been carried out in memory only, so the database still contains the original value. An assumption that is not at all covered by the JPA specification!

Contents of the Persistence Context – The Methods `contains` and `clear`

The `contains` method allows you to check whether a particular entity instance is currently managed by the entity manager. Note that this check is based on object identity, not on the primary key value or the entity's `equals` method. See the following example, where we assume that the entire code is executed within the scope of a transaction:

```
// get a managed entity
Employee managed = em.find(Employee.class, new Integer(10));
// create another instance with the same primary key
Employee notManaged = new Employee(10);

assert em.contains(managed);
assert !em.contains(notManaged); // not identical to managed version!

em.remove(managed);
assert !em.contains(managed); // not managed any more after remove
```

The `clear` method is straightforward – it empties the persistence context. All entities that have been managed before are afterwards detached. Note that clear operates on the entire persistence context – there is no way to remove just a single entity from the control of the entity manager.

Clear the Persistence Context Only When in a Synchronized State

Although clearing the persistence context is a simple operation, there is a potential pitfall when you do so while a transaction is active. The persistence context is cleared immediately, whether or not the managed entities have already been synchronized with the database. Even worse: Unless you explicitly synchronize the persistence context using the `flush` method, you don't know in which state the persistence context actually is: completely synchronized with the database, not yet synchronized at all, or even partially synchronized – the database may contain completely inconsistent data! If you cleared the persistence context in such a state and committed the transaction later, you would have destroyed your data.

To avoid that, clear the persistence context only when you are sure that it is synchronized with the database, that is, within an active transaction only immediately after flush.

In practice, the `clear` method is hardly ever used within the scope of a transaction. It is far more common to use it outside of a transaction to clear an extended persistence context.

Lifecycle of the Entity Manager – The Methods `close` and `isOpen`

The lifecycle of an application-managed entity manager is fully controlled by the application. The application instantiates such an entity manager using the `EntityManagerFactory` interface, uses the entity manager as desired, and finally closes it explicitly using the `close` method. On a container-managed entity manager, however, the `close` method throws an exception.

Correspondingly, the `isOpen` method allows you to check the lifecycle status of the entity manager. The method returns `true` as long as the entity manager is open and ready for usage, or `false` if it has already been closed, either by the container if you have a container-managed entity manager or by the application itself. So the `isOpen` method can in fact be used with both types of entity managers.

Concurrency Control – The `lock` Method

The `lock` method brings up a topic that we have left out so far intentionally – the problem of different transactions trying to modify the same data simultaneously. You can avoid such concurrent data modifications in JPA (basically by preventing all but one of the involved transactions to commit their changes), but we believe it is better to discuss the topic thoroughly in an article of its own.

Transaction-Related Methods – `getTransaction` and `joinTransaction`

If the persistence unit has been configured for resource-local transactions, the `getTransaction` method of the entity manager returns the `EntityTransaction`, which you can use to begin and commit or rollback a resource-local transaction.

Although an application-managed entity manager is most-often used in Java SE applications together with resource-local transactions, it can also be employed within a Java EE application. An application-managed entity manager can even interoperate with JTA transactions, with just slightly more effort for the application.

The entity manager has to listen for the JTA transaction's synchronization events, which notify the entity manager of the end of the transaction. Thus, the entity manager can flush any changes to the database just before commit, for example. In case of a container-managed entity manager, the container takes care that the entity manager is registered for those transaction events. For an application-managed entity manager, however, the application has to call the `joinTransaction` method of the entity manager explicitly to achieve that.

Other Methods – `getReference` and `getDelegate`

The `getReference` method is very similar to the basic `find` operation and allows you to retrieve an entity via its primary key. It is intended for situations where you need a managed entity instance, but you don't access any of its data, besides potentially the entity's primary key. In the discussion of the `remove` method, we have already encountered such a case:

```
// get a managed entity
Employee employee = em.find(Employee.class, new Integer(10));
if (employee != null) {
    em.remove(employee);
}
```

In this example, we are not interested in the entity's data at all. We call the `find` method only to get a managed entity instance, which we then use as input to the `remove` method. An alternative approach would be therefore to call the `getReference` method instead:

```
// get a managed entity
Employee employee = em.getReference(Employee.class, new Integer(10));
em.remove(employee);
```

By using the `getReference` method, you indicate to the JPA provider that you are not interested in the entity's data. The JPA implementation may either ignore that hint and return an entirely loaded entity, or it may choose to return a version of the entity where the data fields, that is anything besides the primary key, are not yet loaded. However, if the JPA provider decides for the latter, it must be guaranteed that you can still access the entity's data as long as the entity is managed (although it is the typical use case for `getReference` that you don't). The JPA implementation must load the data automatically and transparently on demand, i.e. at the time when you access the data rather than when calling the `getReference` method. That mechanism is often called *lazy loading*. Note that lazy loading of data is only possible as long as the entity is managed, with a transaction-scoped persistence context therefore only within an active transaction. If the entity is not or no longer managed, you will get an exception.

Note also that `find` and `getReference` behave differently if the referred to entity does not exist. While `find` returns null, `getReference` throws an exception. The JPA provider may choose when to do so: either immediately as a result of the `getReference` call, or later when the entity's data is actually accessed.

With SAP JPA 1.0, the `getReference` method returns an entity with loaded data, identical to the result of the `find` method. If the entity does not exist, the `getReference` method throws an exception immediately.

The `getDelegate` method provides access to the vendor specific implementation of the `EntityManager` interface. The method returns an object that the user can cast to a vendor-proprietary interface. Thus, the application may combine the standardized JPA programming model with any additional functionality that the particular JPA implementation might offer. The `getDelegate` method is particularly useful for JPA implementations that have been implemented on top of existing persistence frameworks like Hibernate, for example.

In SAP JPA 1.0, the `getDelegate` method is not implemented and throws an `UnsupportedOperationException`.

Related Content

- Getting Started with Java Persistence API and SAP JPA 1.0

  The first article in the series of articles on JPA gives an introduction to the fundamental concepts of JPA and illustrates the programming model by means of a small sample application.

- SAP NetWeaver Application Server, Java EE 5 Edition

  The download package includes: Java EE 5-compliant SAP NetWeaver Application Server, MaxDB 7.6 database, and SAP NetWeaver Developer Studio.

- Java Platform, Enterprise Edition 5 at SAP

  The site collects information on the Java EE 5 platform at SAP, such as, for example, documentation, articles, sample applications and the Java EE 5 forum.

Additional Resources

- The Java Persistence API Documentation (javadoc)

Conventions Within this Document

In all code examples within this document, the variable `em` is assumed to be properly initialized with an instance of `javax.persistence.EntityManager`.

Sections marked with a bulb explain details of the Java Persistence API, point out potential pitfalls intrinsic to the specification or give advice how to use the API efficiently.

The JPA specification leaves certain aspects of the behavior deliberately non-standardized to allow different vendors to choose a suitable implementation. Sections marked with the SAP logo describe how SAP JPA 1.0, the JPA implementation provided with SAP NetWeaver Application Server, Java EE 5 Edition, behaves in such cases.

***Note**: We provide these implementation details for your information only. By no means should your application logic rely on the described behavior. SAP feels free to optimize or revise their implementation choice later on at any point in time.*

Copyright